Support for Flexibility and User Control of Worksharing in OpenMP

Barbara M. Chapman, Lei Huang
University of Houston
<{chapman,leihuang}@cs.uh.edu>

Gabriele Jost
Sun Microsystems, Inc.
<gabriele.jost@sun.com>

Haoqiang Jin
NASA Ames Research Center
<hjin@nas.nasa.gov>

Bronis R. de Supinski
Lawrence Livermore National Laboratory
<supinski@llnl.gov>

NAS Technical Report NAS-05-015, October 2005

Abstract

The set of features in the current OpenMP specification provides essential functionality that was selected mostly from existing shared-memory parallel application programming interfaces (APIs). The requirements of current and emerging architectures, operating systems, and applications prompt us to evaluate and choose new features from many different proposals. In this paper, we describe two application development experiences that exposed problems with expressivity and performance as a result of limitations in the current OpenMP specification. We then propose mechanisms to overcome these limitations, including the notation of thread subteams and thread topology. Our goal is to identify language features that help improve application performance while preserving ease of programming.

1 Introduction

OpenMP is due for an upgrade. The set of features in existing specifications of the API provides essential functionality that was selected mostly from existing shared-memory parallel APIs. We now need to evaluate and choose new features from many different proposals, and in accordance with the requirements of current and emerging architectures, operating systems, and applications. In this paper, we describe two application development experiences that exposed problems with expressivity and performance as a result of limitations in the current OpenMP specification. We then propose mechanisms to overcome these limitations. Our goal is to identify language features that help improve application performance while preserving ease of programming.

While parallelizing these applications with OpenMP, our inability to control the assignment of work to subsets of threads in the current thread team, and to orchestrate the work of different threads, artificially limited the performance we could achieve. To overcome the first difficulty, we propose a new clause for worksharing constructs that assigns the work to a subteam of the existing threads and notation for naming a subteam. Further, we introduce the notion of a topology, which gives a subteam a shape, and library routines to retrieve the number of threads executing a construct, as well as a thread-id based on a named subteam and any reshaping applied to it. We also propose new constructs for posting and waiting for events to support improved work coordination between threads. There is still much to be explored here; this paper gives a first look at the direction our experiences suggest.

Our proposed mechanisms ultimately provide a means to better control work distribution. In many cases, work distribution control provides the same effect as data distribution control. However, work distribution is a more general concept that is likely to improve performance on disparate platforms while many inherently architecture-dependent issues remain with attempts to control data distribution, such as distribution granularity. The code examples and parallelization challenges described here come from two quite different areas.

The first example is based on difficulties encountered while creating an easy-to-maintain OpenMP version of an industrial seismic data processing application for execution on shared-memory processors (SMPs) with hyperthreading. The second example comes from experiences gained while working to create scalable scientific applications on a large distributed shared-memory platform. We outline each of these problems and our preferred strategy for overcoming them in the next two sections. Then, we discuss related work briefly before summarizing our findings.

2 Thread Subteams

Our first example is based on a seismic data processing and interpretation software. We will first describe the problem, and then introduce our solution of thread subteams to overcome the problem.

2.1 The Example of Seismic Data Processing

The example is based on commercial seismic data processing and interpretation software, Kingdom Suite from Seismic Micro-Technology, Inc. Kingdom Suite is an integrated geosciences interpretation software package for Windows systems used by the energy industry in the search for oil. Our OpenMP implementation was applied to TracePak, an I/O-intensive module of Kingdom Suite to analyze and process two-dimensional (2-D) and three-dimensional (3-D) post-stack seismic data [9]. TracePak offers a variety of filters, spectrum manipulations, attribute computations, and other signal operations to allow for a detailed analysis of selected traces. The goal of our effort was to create a parallel program for execution on Windows-based SMPs with hyperthreading enabled. An essential requirement was that the parallel code be as close as possible to the original sequential code to simplify its maintenance. Our findings are based on experiments and analyses of execution behavior carried out on an HP workstation with Dual Hyperthreading-enabled CPUs.

The structure of the sequential program is shown in Fig. 1. Several different seismic data processing functions, mostly FFT-related operations, are invoked based on options selected by the user. This code iteratively reads data from an input file, processes it using different transform functions in the specified order, and then writes the results to an output file. The amount of seismic data typically handled in a job is quite large, ranging from 100MB to 100GB, and reading and writing consume considerable time.

```
1.  for (i=0; i<N; i++) {
2.     ReadFromFile(i,...);
3.     for (j=0; j<ProcessingNum; j++)
4.        for (k=0; k<M; k++) {
5.           ProcessData(); //processing involves several
                            //different seismic functions
6.        }
7.     WriteResultsToFile(i);
8.  }
```

Figure 1: A sequential pseudo-code fragment for seismic data processing

Since OpenMP does not support parallel I/O, we decided that the best strategy to parallelize the code of Fig. 1 is to keep the I/O operations (lines 2 and 7) sequential while overlapping them with the parallelized computation (line 5), as illustrated by the timeline view in Fig. 2. A simple and straightforward way to parallelize the computation is to enclose the innermost loop (k-loop) between threads in an "omp parallel for" directive. Note that it is not feasible to parallelize the outer loop (j-loop) since there is a dependence

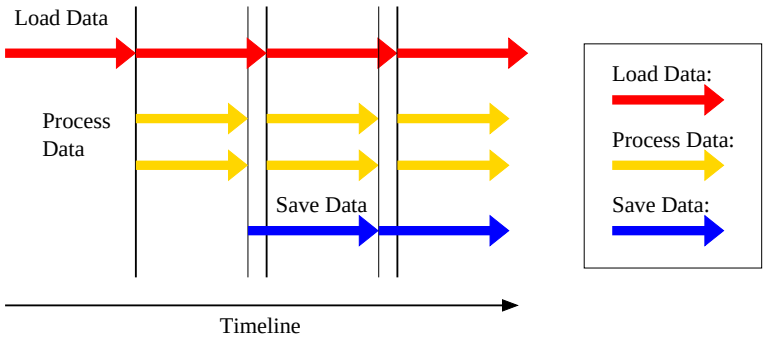

Figure 2: Overlapping I/O with computation in the parallel seismic program

between the seismic data processing functions. However, we will not achieve good scalability by only parallelizing the computation in this manner, since it does not overlap the computation and I/O, and moreover, frequently entering and leaving parallel regions will degrade performance. We need to create a parallel region that enloses the entire loop nest in order to achieve our goal. We show such a version of our code in Fig. 3.

Here, we first preload the data needed for the first iteration of the i-loop (line 4). Then, we use "omp single nowait" and "omp for schedule(dynamic)" so that one thread reads the data for the next iteration and the other threads will proceed to begin computing the j loop (line 11 of Fig. 3). Another thread writes to an output file after the results are ready. When the threads performing I/O complete their work, they may share in the remaining computation under the dynamic scheduling.

```
1.  #pragma omp parallel
2.  {   #pragma omp single
3.      {   //preload data to be used in the first iteration of the i-loop in line 6
4.          ReadFromFile(0,...);
5.      }
6.      for (i=0; i<N; i++) {
7.          #pragma omp single nowait
8.          {   //preload the data for next iteration of the i-loop
9.              ReadFromFile(i+1...);
10.         }
11.         for (j=0; j< ProcessingNum; j++)
12.             #pragma omp for schedule(dynamic)
13.             for(k=0; k<M; k++) {
14.                 ProcessData(); //user configurable data processing functions
15.             } //there is a barrier here
16.         #pragma omp single nowait
17.         {
18.             WriteResultsToFile(i);
19.         }
20.     }
21. }
```

Figure 3: The OpenMP code for seismic data processing kernel

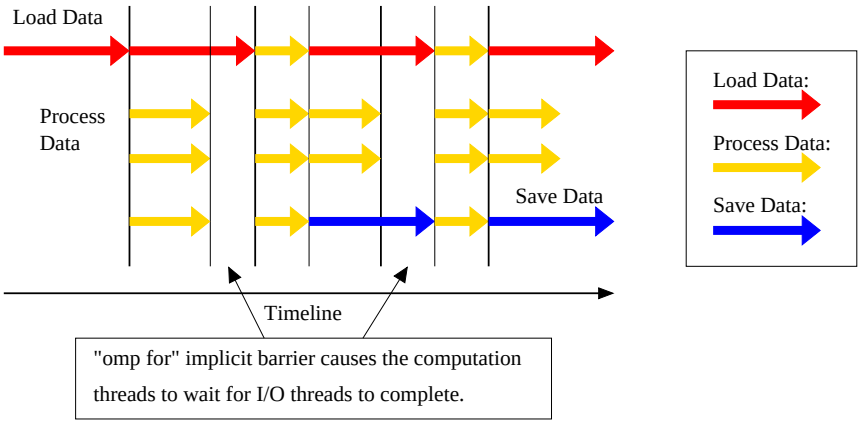

Figure 4: Execution behavior of OpenMP seismic code from Fig. 3

The innermost, work-shared loop includes an implicit barrier at its end. Unfortunately, we cannot simply remove it since the data processing functions must follow a specific sequential order: each iteration uses results from the previous one. So although plenty of computation remains, the computing threads must wait at the implicit global barrier from the time it is first reached until the I/O has completed, as shown in Fig. 4. Thus this approach does not fully overlap I/O operations and computation. A significant reprogramming effort would be needed to overcome the problem. For example, exchanging the order of the loops in the nest would, if at all possible, require a complete rewrite. However, ease-of-use and maintenance motivated our use of OpenMP. A parallelization strategy that requires major code reorganization is not acceptable for such a large commerical application. This also rules out a lower-level programming style that coordinates threads explicitly and does not use directives.

2.2 Performance Improvement

In a normal run, the ratio of I/O and computation is about 1.2:1, where the I/O takes a little more time than the computation does. Thus, if we could avoid including the I/O threads in the barrier operation, it should be possible to overlap I/O with the computation. Since the current OpenMP specification does not easily support this behavior, we combined OpenMP with Windows threads for reading and writing files and achieved much greater overlap. Fig. 5 shows results on an HP XW8200 with dual Xeon 3.4 GHz CPUs, 1MB L2 cache, 3GB memory, Intel extended memory 64, and hyperthreading enabled. The compiler used was Microsoft Visual C++ in Visual Studio 2005 Beta 2 with OpenMP support. The hybrid version improved the performance by 25% on four threads over the standard OpenMP version.

To achieve similar results with pure OpenMP, we require mechanisms to separate the computational threads from the data handling threads, and to synchronize their activities in a proper fashion. We could achieve the separation with three parallel sections: read, write, and computation. The computation section would create a nested parallel region and share the work among its threads. To synchronize properly, we either prefetch data in the previous iteration, as in the code of Fig. 3, or use critical regions and arrays of variables. Still, each iteration of the outer i-loop requires a new parallel region if we are to retain the sequential program structure.

2.3 Thread Subteam as a Solution

Nested parallelism can dynamically create, exploit and terminate teams of threads and is well-suited to codes with needs that change over time. Our code structure is static. The relative amount of data and computation

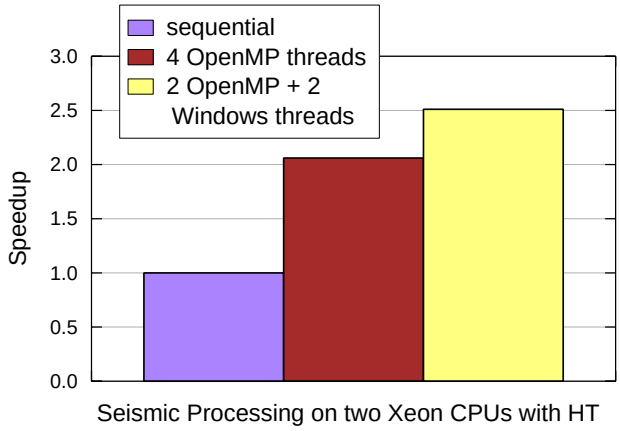

Figure 5: Performance comparison: OpenMP vs. hybrid OpenMP and Windows API codes

does not vary, and we expect the number of participating threads and their roles to remain the same. Nested parallelism is more powerful than we require. Thus, we propose a simpler mechanism that provides the desired separation and the required synchronization. Specifically, we define subteams of threads, similar to subcommunicators in MPI. These subteams allow us to bind the execution of a worksharing construct or a barrier construct to a subset of threads in the current team. Only the threads in the subteam participate in its work, including any barrier operations encountered. To synchronize the actions of multiple subteams, we may use existing OpenMP constructs and take advantage of the shared memory.

To realize this idea, we define an additional "onthreads" clause that may be applied to worksharing and barrier directives. This clause permits us to specify the target of a worksharing directive to work on a *subteam* of threads. It may refer only to existing threads; in other words, in contrast to nested parallelism, this does not create any new threads. When the subteam of threads specified is not the entire current team, it restricts participation in the associated work to the specified members. In particular, implicit and explicit barriers within the code it encloses do not block threads that are not part of the subteam. This clause would require minimal change to the current specification since the entire team will share in the work by default. Alternatively, we can define an "onthreads" directive that could enclose an arbitrary structured block of code within a parallel region. Work in the block would be carried out by the specified subteam of threads.

Using the thread subteam notation, we can rewrite the example code in Fig. 3 to that in Fig. 6. Line 5 and line 14 use the "onthreads" clause to limit the I/O to individual thread, while line 7 defines a subteam of threads to process data. The integer expression(s) in brackets uses OpenMP's thread-ids and array section notation to specify the desired subset of threads. A general notation for specifying thread range would be "(f:l:s)" where f is the first thread-id, l is the last thread-id, and s is the step. The implicit barrier at line 12 applies only to the threads defined in the subteam from line 7.

Additional syntax could enable the programmer to name these subsets. New run-time library routines would be provided to get the number of threads of a named subteam and to retrieve a subteam-internal consecutive thread number. A programmer might also want to permute the order of threads in a subteam to specify schedules that enforce a certain work distribution, that is, the assignment of specific work (chunks), which would support data reuse by that thread. Although none of these (except possibly the library routines) are essential, they would greatly increase the expressive power of this construct. Interactions between subteams could be made explicit by providing new notation for communication between subteams of threads. This might help a programmer reason about the structure of this communication and avoid programming errors such as deadlock. The same construct might also enable point-wise synchronization between threads in a single subteam, e.g., when a dependence between two variable references in a parallel loop would oth-

```
1.  #pragma omp parallel
2.  {  #pragma omp single
3.        ReadFromFile(0,...); //preloads data for first iteration of i-loop
4.     for (i=0; i<N; i++) {
5.        #pragma omp single onthreads(0)
6.          ReadFromFile(i+1...); //preload data for next iter. of i-loop
7.        #pragma omp onthreads ( 2:omp_get_num_threads()-1 )
8.        for (j=0; j< ProcessingNum; j++)
9.          #pragma omp for schedule(dynamic)
10.         for (k=0; k<M; k++) {
11.            ProcessData(); //user configurable data processing functions
12.         } //here is the group-internal barrier
13.       #pragma omp barrier  //this ensures we are ready for next iter.
14.       #pragma omp single onthreads(1)
15.         WriteResultsToFile(i);
16.    }
17. }
```

Figure 6: OpenMP code with the "onthreads" directive for seismic data processing kernel

erwise require a barrier. In the code fragment reproduced in Fig. 7, a post-wait notation does this succinctly and we have named the thread team, whose order is a permutation of the original thread numbers (used here only to illustrate the concept since the work distribution does not provide any data reuse).

```
#pragma omp parallel
{
   #pragma omp team CompthreadsReordered = threads(omp_get_num_threads()-1:2:-1)
   for (i = 0; i < N; i++) { //executed by all threads
       #pragma omp single onthreads(0)
       {  ReadFromFile(i);
          #pragma omp post (dataready[i])  //signals reading is complete
       } //thread(0) independently does this reading and posting
       ........
       #pragma omp on CompthreadsReordered
       {  //subteam starts to work
          #pragma omp wait (dataready[i])  //after data is ready
```

Figure 7: Excerpt from OpenMP code with named subteam and post/wait

The ability to divide work among subteams of threads, and thus to have different subteams working concurrently and independently, seems to be a fairly natural extension to the current API and it has a variety of potential uses. It would likely simplify the use of OpenMP within third party libraries. It also enables the specification of multi-disciplinary code ensembles and permits components written in traditional programming languages to interact without the need to provide external file-based interactions. It supports the simpler case of multilevel parallelism with a fixed team of threads without the extra overheads and burden of nested parallelism.

3 Worksharing and Synchronization Across Loop Nests

Scientific and engineering computations must exploit large numbers of threads, not only in emerging, very large shared-memory systems, but also in smaller SMPs with chip multiprocessors. More attention must be paid to achieving scalable code. Two of the authors previously proposed a set of language features to enable the parallelization of multiple levels of loop nests [6]. These features allow specification of an appropriate execution schedule and the assignment of threads to loop levels, as well as additional synchronization that enables a pipelined execution scheme in the LU benchmark from the NAS Parallel Benchmarks [1]. They reported on a prototype implementation. This work was motivated by the fact that applications of interest did not scale to the desired CPU count, even though there was sufficient inherent parallelism.

3.1 The LU Example

The LU application benchmark uses the symmetric successive over-relaxation (SSOR) method to solve a seven band block-diagonal system. Figure 8 illustrates the lower triangular phase of the LU kernel example. References to values of elements of array v in line 4 of the code create dependences between loop iterations that prevent straightforward parallelization. However, a wave-front or a pipelined technique can enable considerable levels of parallelism to be exploited, since the value of an element of v can be computed once the new values are available from the previous iteration in each of the three dimensions.

```
1.  do k = 2, nz
2.    do j = 2, ny
3.      do i = 2, nx
4.        v(i,j,k) = v(i,j,k) + a*(v(i-1,j,k) +
  &                  b*v(i,j-1,k) + c*v(i,j,k-1)
5.        .  .  .
6.      enddo
7.    enddo
8.  enddo
```

Figure 8: The LU computational kernel

A wave-front restructuring of the code reveals parallelism that can be expressed with the standard OpenMP parallel directive to update points on a diagonal plane concurrently. However, this method suffers from poor cache utilization. A pipelined approach, in which data are partitioned as blocks in selected dimensions, usually gives better cache performance. We illustrate the differences between wave-front and pipelined parallelism in Fig. 9. Expression of the parallelism in two dimensions would reduce the cost of pipeline startup and shutdown, and support good cache performance for this kernel. However, OpenMP currently can only successfully exploit parallelism in one dimension. Using OpenMP parallelization in multiple dimensions would require nested parallelism, which results in multiple one-dimensional pipelines and incurs high overheads [5].

3.2 Thread Topology

We introduce the notion of a thread topology to support pipelined algorithms. A thread topology does not create new threads; instead, it reshapes the thread (sub)team and associates a new naming scheme with existing threads (whose default names are natural numbers starting from zero). We can then use the topology to specify a variety of new schedules for worksharing directives. Our syntax requires the programmer to

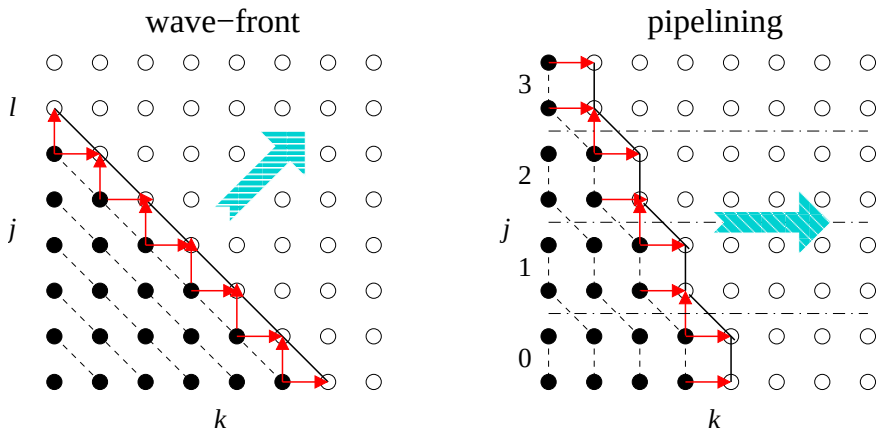

Figure 9: Schematic illustration of the wave-front and pipelined algorithms. j and k are the two data dimensions. l in the left panel indicates a diagonal plane. Numbers in the right panel indicate partitioned data blocks mapped to different threads.

provide the number of dimensions in the topology and the coordinates within each dimension. To use this properly, we will need a default strategy for mapping the linearly numbered threads to a Cartesian grid. The basic syntax of specifying a topology is:

```
!$omp topology name(ndim, start, stop, stride, fixedorder)
```

where `name` defines a name of the topology. The `ndim` argument specifies the number of dimensions in the topology. The arguments `start`, `stop`, and `stride` are arrays with one entry per dimension to specify the topological shape. `fixedorder` is a Boolean variable that tells the compiler whether or not the default strategy for associating these threads with the linear thread numbers must be applied. If not, the system can choose any mapping of the threads to the topology. For example, if 16 threads have been created, the directive

```
!$omp topology mygrid(3, start, stop, stride, .false.)
```

can be used to reshape threads into a $4 \times 2 \times 2$ grid with coordinates from (0,0,0) to (3,1,1) or any other numbering scheme we wish to define that uses 16 threads. In this case, `start`, `stride`, and `stop` are 3-D arrays.

Once a topology is defined, we need to associate it with a worksharing construct using the "onthreads" clause. We use standard section notation to specify the target of the worksharing directive in each topological grid dimension. We use ":" to denote the entire dimension of an array (so worksharing to "onthreads(:)" would map the computation to all threads). Dimensions not involved in the worksharing are marked via a dummy "*" and the computation is replicated in those dimensions.

The use of this notation is illustrated in Fig. 10 for the LU computational kernel. We introduce a 2-D logical grid of threads with the same number of threads in each dimension. We now want to use this and our thread subteam clause to specify the target of our worksharing construct. Thus, we map the iterations of two different loops to threads using the convenience of a 2-D naming scheme, our grid topology. We realize this using two worksharing constructs (this notation does not conform to current OpenMP rules). The 2-D topology is used to distribute the work in the i and j loop nests among threads. The first worksharing directive spreads iterations of the j-loop among the first dimension of the thread topology; the second directive maps iterations of the i-loop among thread-ids in the second dimension so that each thread of the grid has its own portion of the work of the loop nest.

```
     mystart(1) = 1; mystart(2) = 1 ...   ! assign values to mystart (:) and mystop(:)
    !$omp parallel
    !$omp topology grid(2,mystart,mystop,mystride, .true.)
                                          ! arrange threads logically into a square called grid
     iam1 = omp_get_coord(grid,1)
     iam2 = omp_get_coord(grid,2) ! my coords in grid
1.   do k = 2, nz
       !$omp wait grid (iam1-1,iam2)  ! wait for thread below to complete its portion
       !$omp wait grid (iam1,iam2-1)  ! wait for thread on left to complete its portion
       !$omp do onthreads(grid(:,*))  ! share out to first dimension of grid
2.     do j = 2, ny
         !$omp do onthreads(grid(*,:))    ! share out to second dimension of grid
3.       do i = 2, nx
4.         v(i,j,k) = v(i,j,k) + a*v(i-1,j,k) + b*v(i,j-1,k) +
   &                  c*v(i,j,k-1)
5.         .  .  .
6.       enddo
         !$omp end do nowait
7.     enddo
       !$omp end do nowait
       !$omp post grid(iam1,iam2+1)  ! indicate to thread on right that it is ready
       !$omp post grid(iam1+1,iam2)  ! indicate to thread above that it is ready
8.   enddo
```

Figure 10: The multilevel LU computational kernel using thread topology

Finally, we need a mechanism to define synchronization between threads in a topology. We cannot use existing features of OpenMP, since the interaction required is not between iterations but threads. This is achieved here using `post` and `wait` directives with thread-ids, in this case 2-D ids defined by our topology. In our example, dependences require each thread of the topology to wait for its neighbors to the left and below it to finish their computation. Once its work is done, a thread signals its neighbors to the right and above that they can continue. We have used a notation that seemed easiest in this case: it specifies threads based on their relative position in the grid. For flexible synchronization, threads will need to be able to signal arbitrary threads in the topology. To do so, each thread needs to be able to access its (possibly multidimensional) logical id. The ability to synchronize between threads is very important for implementing the pipelined approach in the LU algorithm. In general, it enables loosely synchronous algorithms. One of the hardest parts of a translation of this kind of code is avoiding false sharing of cached data. As suggested by Liu et al. [8], creating private copies of the "local" portions of arrays that can be updated in the loop nest, and then copying the results back to the global shared array is one possible way to do so.

4 Related Work

The NanosCompiler team at the European Center for Parallelism in Barcelona (CEPBA) has proposed the creation of groups of threads in association with parallel regions [2, 3, 4]. Their notation permits the user to specify the number of independent teams of threads that will be created. Since these thread groups are associated with the parallel region, additional notation is required to assign work to the individual groups.

They also propose extensions to easily express the precedence relations that originate pipelined computations. These extensions are also valid in the scope of nested parallelism and are based on the ability to name worksharing constructs and to specify a predecessor-successor relationship between worksharing constructs to support synchronization. In contrast, we specify the target of a worksharing directive, i.e., the group of threads that share the work. Further, our topology renames threads to give the target a multidimensional structure that simplifies specifying the desired target sets. The predecessor-successor relationship imposes an orderings in terms of iterations or sections, it imposes an ordering on the actions of threads, so its meaning is less intuitive. We refer to threads explicitly and, thus, avoid this problem. Furthermore, our proposal does not involve nested parallelism and the associated overhead and restrictions.

There have been a variety of proposals for multilevel loop parallelism. The SGI compiler for the Origin platforms [7] provides limited support by accepting the SGI NEST clause on the OMP DO directive. The NEST clause requires at least two variables as arguments to identify indices of subsequent DO-loops. The identified loops must be perfectly nested. No code is allowed between the identified DO statements and the corresponding END DO statements. The nest clause on the OMP DO directive informs the compiler that the entire set of iterations across the identified loops can be executed in parallel. The compiler can then linearize the execution of the loop iteration and divide them among the available single level of threads.

Intel has proposed a wavefront directive to enable wavefront execution schema. Although this might sometimes be appropriate, we expect that it will be hard to achieve good data locality in most cases. Our proposal explicitly enables control of work distribution and, thus, enables the expression of data locality.

5 Conclusions

OpenMP is a shared-memory programming API that offers the promise of performance and ease of use. It is currently deployed on both small and large SMPs, including systems with distributed global memory and new platforms with hyperthreading. It seems possible that the judicious addition of language features that increase the power of expressivity might also improve the achievable performance of a variety of OpenMP codes. In this paper, we introduced a unified notation for sharing work among subteams of threads and for flexibly executing multiple levels of loop nests in parallel, including specifying the synchronization required between the work performed by different threads. We anticipate the addition of a few routines to the runtime library that return the number of threads in a subteam and the coordinates of a thread in a topology. This work could be extended in a variety of ways. Additional notation could be defined to name groups of threads, to more easily specify multiple orderings, to handle synchronization between threads in codes with different kinds of boundary conditions, and perhaps to help a compiler and runtime system find an appropriate data layout.

Acknowledgment

This work was partly performed by B. Chapman when she was visiting NASA Ames Research Center in May 2005. We wish to thank Seismic Micro-Technology Inc. for their help in understanding Kingdom Suite and Holly Amundson from the NAS Publications and Media group for her useful edit comments.

References

[1] D. Bailey, T. Harris, W. Saphir, R. Van der Wijngaart, A. Woo, and M. Yarrow, “The NAS Parallel Benchmarks 2.0,” RNR-95-020, NASA Ames Research Center, 1995.

[2] M. Gonzalez, E. Ayguade, X. Martorell, J. Labarta, N. Navarro, and J. Oliver. “NanosCompiler: Supporting Flexible Multilevel Parallelism in OpenMP.” Concurrency: Practice and Experience. Special issue on OpenMP, vol. 12, no. 12, pp. 1205-1218, October 2000.

[3] M. Gonzalez, E. Ayguade, X. Martorell and J. Labarta. “Defining and Supporting Pipelined Executions in OpenMP.” 2nd International Workshop on OpenMP Applications and Tools, July 2001.

[4] M. Gonzalez, J. Oliver, X. Martorell, E. Ayguade, J. Labarta, and N. Navarro. “OpenMP Extensions for Thread Groups and Their Run-time Support.” 13th International Workshop on Languages and Compilers for Parallel Computing (LCPC’2000), New York (USA), pp. 317-331, August 2000.

[5] H. Jin, G. Jost, J. Yan, E. Ayguade, M. Gonzalez, and X. Martorell, “Automatic Multilevel Parallelization Using OpenMP,” Scientific Programming, Vol. 11, No. 2, pp. 177-190, 2003.

[6] H. Jin and G. Jost. “Support of Multidimensional Parallelism in the OpenMP Programming Model,” WOMPEI2003, Tokyo, Japan, October 2003, in the Proceedings of the International Symposium on High Performance Computing (ISHPC-V).

[7] MIPSPro 7 Fortran 90 Commands and Directives Reference Manual 007-3696-03. http://techpubs.sgi.com/.

[8] Z. Liu, B. Chapman, Y. Wen, L. Huang and O. Hernandez. “Analyses and Optimizations for the Translation of OpenMP Codes into SPMD Style,” Proc. WOMPAT 03, LNCS 2716, 26-41, Springer Verlag, 2003.

[9] TracePak Module, www.seismicmicro.com/Prod_Geo.htm.